Recherche de cycles

Damien Reimert
ENS de Lyon
damien.reimertvasconcellos@ens-lyon.fr

11 décembre 2010

Table des matières

1 Introduction

La recherche de cycles est un problème récurrent en algorithmique des graphes, avec différentes variantes suivant les propriétés recherchées sur ces cycles (par exemple des propriétés de longueur). Les applications sont très nombreuses, avec des problèmes comme celui du "postiers" ou "voyageurs de commerce".

Je vais commencer par donner quelques définitions sur les graphes et les cycles. Je vais ensuite vous présenter quelques complexité avant de présenter 2 algorithmes et de conclure.

2 Définition

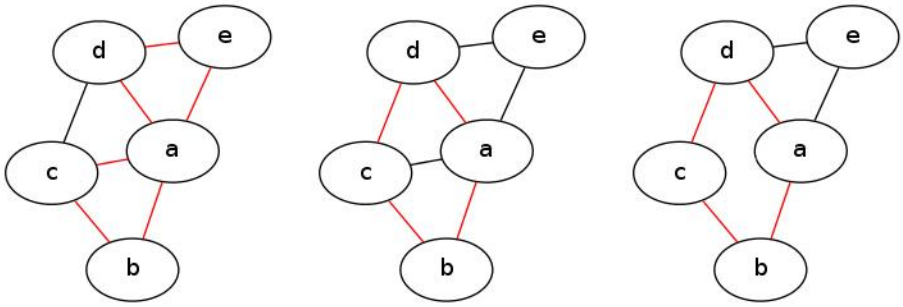
Soit $G = (V, E)$ un graphe où V est l'ensemble des sommets et E l'ensemble des arêtes. Si les arêtes sont non-orientées alors on dit que G est un graphe non-orienté. Si les arêtes sont orientées, on les appel alors arcs et on dit que G est un graphe orienté ou que G est un digraphe.

On notera n le nombre de sommet du graphe avec $n = \#V$ et m le nombre d'arêtes du graphe avec $m = \#E$.

Dans une graphe non-orienté, une chaîne de x à y est une liste de sommets commençant par x et terminant par y telle qu'il existe une arête entre chaque paire de sommets consécutifs. La longueur d'une chaîne est le nombre d'arêtes parcourues. Dans un graphe orienté, une chaîne sera appelée un chemin.

Dans un graphe non-orienté (graphe orienté), un cycle (circuit) est une chaîne (chemin) dont les 2 extrémités sont identiques. Par abus de langage, parfois, on ne fera pas la différence entre cycle et circuit.

On parle de cycle (circuit) standard quand l'on ne passe pas 2 fois par la même arête (arc) et de cycle (circuit) élémentaire (*simple cycle* en anglais) quand l'on ne passe pas 2 fois par le même sommet. Dans la figure 1 (a), le cycle $C = (a, b, c, a, d, e, a)$ (en rouge sur le graphe) est un cycle standard car on ne passe que une fois par arête mais n'est pas un cycle élémentaire car on passe plusieurs fois par le sommet a. La figure 1 (b), montre un cycle élémentaire $C_1 = (a, b, c, d, a)$ (en rouge). C_1 n'est pas un trou car il y a une corde (a, c). Dans la figure 1 (c), le cycle C_1 est un trou car il n'y a plus de corde.



(a) (b) (c)

FIGURE 1 – Differents types de cycle

Si un cycle est de longueur paire (contient un nombre paire d'arête) on parlera de cycle paire. De même, on parlera de cycle impaire pour des cycles de longueur impaire.

On parlera de cycle sans corde ou trou pour désigner un cycle élémentaire qui n'a pas de corde, c.à.d., qu'il n'y a pas d'arête hors du cycle qui relie 2 sommets du cycle. Un anti-trou sera le complémentaire d'un trou. Les cycles sans corde ne sont que des chaînes sans cordes dont le début et la fin sont identiques. Un chemin sans corde est donc un chemin tel qu'il n'y a pas d'arêtes n'appartenant pas au chemin qui relient deux sommets du chemin.

Autre notion sur les cycles, les cycles minimums. Un cycle est minimum si ça longueur est minimal, c.à.d., que c'est le plus petit cycle du graphe. Par exemple, dans la figure 1 le cycle minimum est $C = \{d, e, d\}$.

Un graphe G est k-dégénéré s'il est possible d'ordonner (numéroter) ses sommets de telle façon que tout sommet a au plus k voisins le précédant dans cet ordre. Le graphe de la figure 2 est 3-dégénéré, chaque sommet a au plus 3 voisins qui lui sont inférieur. Moins un graphe est dégénéré (plus k est petit) plus le graphe a de chance d'être d'être clairsemé. On dit qu'un graphe est clairsemé quand son nombre d'arête est faible par rapport à son nombre de sommet. Si il a un nombre important d'arête pour son nombre de sommet alors on dit le graphe dense.

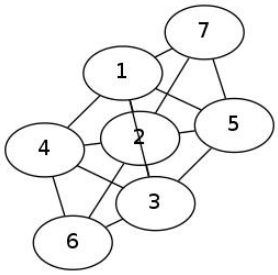


FIGURE 2 – Graphe k-dégénéré

3 Complexité

Les tables 1 et 2 présentent des complexités pour différents types de cycle en fonction de se qui est recherché.

	Cycle standard	Cycle élémentaire	Cycle sans corde
Longueur minimum	/	$O(n^2)$ [5]	
Longueur k avec k fixé	$O(n^{2-2/(k+1)})$ [1]	$O(n^{2-2/(k+1)})$ [1]	$O(n^k)$ [4]
Longueur paire	$O(n^{2-2/k})$ [1]	$O(n^{2-2/k})$ [1]	$O(n^{15})$ [2]
Longueur impaire	$O(n^{2-2/(k+1)})$ [1]	$O(n^{2-2/(k+1)})$ [1]	

TABLE 1 – Divers complexité dans les graphes non-orientés

	Circuit standard	Circuit élémentaire	Circuit sans corde
Longueur minimum	/	$O(nm)$ [3]	
Longueur k avec k fixé	$O(n^{2-2/(k+1)})$ [1]	$O(n^{2-2/(k+1)})$ [1]	
Longueur paire	$O(n^{2-2/k})$ [1]	$O(n^{2-2/k})$ [1]	
Longueur impaire	$O(n^{2-2/(k+1)})$ [1]	$O(n^{2-2/(k+1)})$ [1]	

TABLE 2 – Divers complexité dans les graphes orientés

Il est à noter que certaines cases n'ont pas de sens. Par exemple, un cycle standard de longueur minimum. Un cycle de longueur minimum est obligatoirement élémentaire sinon il n'est pas de longueur minimum. En effet, si un cycle passe 2 fois par un sommet alors ce "détour" est inutile.

Les tables 1 et 2 donne une idée de la complexité de quelques problèmes classique, mais il y a de nombre cas particuliers où l'on peut encore affiner cette complexité.

Dans les tables 1 et 2, j'ai donné la complexité dans le pire cas sur des graphes quelconques, le seul critère retenu étant de savoir si le graphe est orienté ou non. Mais un certain nombre d'algorithme ont un temps moyen de terminaison inférieur. Par exemple, la recherche de cycle minimum dans un graphe non-orienté a un temps moyen de terminaison en $O(n^2)$ alors que son

pire cas est en $O(nm)$ dans l'article de Itai and Rodeh [3]. Et il en est de même, toujours dasn le même article, pour la recherche de dicircuit minimum avec une complexité dans le pire cas en $O(nm)$ et un temps moyen en $O(n^2 logn)$.

D'autre algorithme ont une complexité qui dépend d'un de leurs paramètres. Par exemple, les algorithmes de recherche de cycle de longueur k. Dans ces algorithmes, plus k est grand plus la complexité est grande.

Il y a aussi des algorithmes qui pour des conditions particulières donne de bien meilleur complexité. Par exemple, pour la recherche de cycle de longueur k. L'article de Alon, Yuster et Zwick [1], nous propose 2 complexités différentes. La première dépend uniquement du nombre de sommet et du k recherché alors que la seconde dépend, en plus, de la dégénérescence du graphe. Le même article, nous propose une complexité en $O(V^\omega)$ avec $\omega < 2.376$ qui est la complexité de la multiplication de matrice, pour compter le nombre de cycle de longueur k.

Pour finir, des algorithmes très spécialisés réussissent à avoir des complexités encore meilleur. Il est par exemple possible de trouver des triangles (des cycles de longueur 3) dans un graphe planaire en $O(n)$.

4 Algorithme

Je vais maintenant vous présenter un algorithme qui détermine si un graphe non-orienté $G = (V, E)$ possède un trou avec une complexité $O(n + m^2)$ (avec $n = \#V$ et $m = \#E$).

4.1 Détection de trou

Je vais vous présenter l'algorithme de détection de trou de Nikolopoulus et Palios [4].

À partir de maintenant, on utilisera la notation P_k pour désigner un chemin sans corde de longueur de k.

L'algorithme repose sur le Lemme 1. Vous pourrez trouver la démonstration de ce lemme dans l'article de Nikolopoulus et Palios [4].

Lemme 1. *Un graphe non-orienté G contient un trou si et seulement si G contient un cycle $C = (u_0, u_1, ..., u_k)$, où $k \geq 4$, tel que $(u_i, u_{i+1}, u_{i+2}, u_{i+3})$ pour tout $i = 0, 1, ..., k-3$, et $(u_{k-2}, u_{k-1}, u_k, u_0)$ sont des P_4 de G.*

L'idée de la preuve est de supposer que G contient au moins un cycle qui remplit les conditions du lemme. On choisit ensuite le cycle minimal parmi ceux-ci et on suppose qu'il contient une corde. Il suffit de montrer ensuite que dans ce cas il n'est pas minimal et qu'il y a contradiction.

L'idée de l'algorithme est d'essayer d'étendre un $P_3 = (a, b, c)$, facilement détectable, en un $P_4 = (a, b, c, d)$. Puis une fois ce P_4 trouvé, de réitérer l'opération sur le $P_3 = (b, c, d)$. Si on ajoute un sommet déjà rencontré alors on a détecté un cycle.

Pour fonctionner l'algorithme utilise un tableau not_in_hole$[(u, v), w]$. Ce tableau sert à ne pas recalculer un P_3 déjà calculé. Les index sont un couple d'une arêtes et d'un sommet et les valeurs sont un booléen . Il est à noter que l'arête (u, v) et l'arête (v, u) correspondent au même index car G est non-orienté. La valeur de not_in_hole$[(u, v), w]$ est 1 (ou vrai) si et seulement si les sommets u, v, w induisent un $P_3 = (u, v, w)$ qui a déjà été calculé et qui ne fait pas parti d'un trou sinon la valeur est 0 (ou faux).

Un second tableau est utilisé in_path$[]$. Ce tableau indique si un sommet à déjà été visité et si il fait partie du trou que l'algorithme essayé de construire. Si c'est le cas, alors la valeur est 1 (ou vrai), sinon elle est de 0 (ou faux).

Le fonctionnement de l'algorithme de détection de trou est donc le suivant :

- Après l'étape d'initialisation (ligne 1 à 2), on cherche tous les $P_3 = (u, v, w)$ potentiel du graphe (ligne 3 à 6).
- Pour chacun d'entre eux, on vérifie qu'il n'ont pas déjà été calculé (condition not_in_hole$[(u, v), w] = 0$ à la ligne 6) et on indique que les sommets u et v font partis du trou que l'algorithme essaie de trouver (ligne 4 et 7).

Algorithm 1 Detection de trou

Require: un graphe connexe non-orienté G.
Ensure: vrai, si G contient un trou ; faux sinon.

```
1: Initialiser les entrées des tableaux not_in_hole[] et in_path[] à 0 ;
2: Calculer la matrice d'adjacence A[] de G ;
3: for each sommet u de G do
4:    in_path[u] ← 1 ;
5:    for each arête vw de G do
6:       if u est adjacent à v et non-adjacent à w et not_in_hole[(u, v), w] = 0 then
7:          in_path[v] ← 1 ;
8:          if process(u, v, w) then
9:             return true ;
10:         end if
11:         in_path[v] ← 0 ;
12:      end if
13:   end for
14:   in_path[u] ← 0 ;
15: end for
16: return false
```

- Si un P_3 n'a effectivement pas encore été calculer, alors on appel la fonction process(a, b, c) (ligne 8) qui va essayé de construire un trou à partir de ce P_3 (process sera détaillé plus loin). Si un trou est détecté, alors l'algorithme s'arrête (ligne 9).
- Dans le cas contraire, on réinitialise in_path[] en retirant u et v (ligne 11 et 14) et on recommence pour le P_3 suivant (ligne 3 et 15).
- Si tous les P_3 ont été testé et qu'aucun trou n'a été trouvé alors il n'y a pas de trou et l'algorithme renvoie faux (ligne 16).

En regardant l'algorithme, il est facile de voir que si process termine et remplit correctement not_in_hole[] alors l'algorithme termine.

Algorithm 2 process(a, b, c)

Require: a, b, c tel que (a, b, c) est un P_3.

```
1: in_path[c] ← 1 ;
2: for each sommet d adjacent à c in G do
3:    if d est adjacent ni à a ni à b in G then
4:       if in_path[d] = 1 then
5:          G a un trou ;
6:          return true ;
7:       else if not_in_hole[(b, c), d] = 0 then
8:          if process(b, c, d) then
9:             return true ;
10:         end if
11:      end if
12:   end if
13: end for
14: in_path[c] ← 0 ;
15: not_in_hole[(a, b), c] ← 1 ;
16: not_in_hole[(c, b), a] ← 1 ;
17: return false
```

Le fonctionnement de la fonction process est le suivant :

- On indique que le sommet c fait parti du trou que l'algorithme essaie de trouver (ligne 1).

– On regarde pour tous les voisins de c (ligne 2) si il y en a un qui permet d'étendre le $P_3 = (a, b, c)$ en un $P_4 = (a, b, c, d)$ (ligne 3).
– Si on en trouve un et qu'il fait déjà parti du trou (ligne 4) alors on a trouvé un trou et la fonction termine en renvoyant vrai (ligne 6).
– Sinon, on regarde si le nouveau $P_3 = (b, c, d)$ a déjà été calculé (ligne 7) et dans le cas contraire, on le calcule en rappelant process dessus (ligne 8). Si la chaîne de récursion trouve un trou (ligne 8) alors la fonction termine est renvoie vrai.
– Si aucun trou n'est trouvé, on essaie pour le voisin suivant de c (ligne 2 et 13).
– Si après avoir testé tous les voisins de c aucun trou n'a été détecté, alors on retire c des sommets possible pour faire parti du trou (ligne 14). Et on indique que les P_3 que l'on peut construire à partir des arêtes (a, b) ou (b, c) ne font pas parti d'un trou.
– On fini par indiquer qu'aucun trou n'a été trouvé en renvoyant faux.

En regardant l'algorithme, il n'est pas évidant de voir qu'elle termine. Intuitivement, la chaîne de récursion termine. En effet, on ne peut pas faire 2 fois le même appel à process sinon cela signifie que l'on a déjà rencontré un des sommets, hors la fonction se serait arrêtée dans ce cas.

4.2 Détection de circuit

Nous allons maintenant voir comment trouver un circuit de longueur minimum. Pour cela, je vais vous présenter l'algorithme de Itai et Rodeh [3].

Nous allons d'abord l'algorithme FRONT qui permet de trouver une borne inférieur sur la longueur du circuit de taille minimum atteignable depuis un sommet v du graphe G.

FRONT réalise un parcours en large partielle de G depuis v niveau par niveau. Si la composante connexe contenant v ne contient pas de circuit alors $k(v)$ sera égale à l'infini. $k(v)$ est égale au dernier niveau dans lequel la recherche a été effectué.

L'idée de l'algorithme est de construire un arbre à partir de v et de s'arrêter lorsque un cycle est détecté.

La borne inférieur sur la longueur du circuit minimum accessible depuis v est $2k(v) + 1$.

Pour fonctionner, l'algorithme utilise une queue (structure de stockage de données fonctionnement en mode FIFO, dans laquelle le premier élément à rentrer est le premier élément à sortir). On utilisera enqueue(u) pour ajouter un élément dans la queue et dequeue pour récupérer la valeur du premier élément et le retirer de la queue.

Algorithm 3 FRONT($v, k, level$)

```
1:  for u ∈ V do
2:      level(u) ← nil ;
3:  end for
4:  level(v) ← 0 ;
5:  enqueue(v) ;
6:  while la queue n'est pas vide do
7:      u ← dequeue ;
8:      for w ∈ A(u) do
9:          if level(w) = nil then
10:             level(w) ← level(u) + 1 ;
11:             enqueue(w) ;
12:         else if level(u) ≤ level(w) then
13:             k(v) ← level(u)
14:             return
15:         end if
16:     end for
17: end while
18: k(v) ← ∞
```

L'algorithme fonctionne de la façon suivante :
- On commence par initialiser le niveau de chaque sommet à nil (ligne 1 à 3).
- On fixe le niveau de v à 0 (ligne 4) et on l'ajoute dans la queue (ligne 5).
- On parcours tout le graphe jusqu'à détecter un cycle ou que tous les sommets accessibles ai été parcouru (ligne 6 à 17).
- Pour cela, on récupère le premier sommet de la file (ligne 7).
- Pour chaque voisin de ce sommet (ligne 8 à 16), si il n'a pas été calculé (ligne 9), c'est qu'il appartient au niveau suivant du niveau courant (ligne 10) et on l'ajoute à la queue (ligne 11).
- Sinon, si le niveau du voisin est supérieur ou égal à celui du sommet (ligne 12), c'est qu'il y a un cycle. Dans ce cas, le niveau courant est le $k(v)$ (ligne 13) et on arrête l'algorithme (ligne 14).
- Si aucun cycle n'est trouvé, alors on fixe le $k(v)$ à l'infini.

L'algorithme parcours, dans le pire cas, la totalité des sommets du graphe, ça complexité est donc $O(n)$.

Grâce à FRONT, on peut maintenant déterminer si le graphe a un cycle et on a une borne inférieur sur sa taille. On va maintenant essayer de trouver la taille exacte de ce cycle.

Quand on exécute FRONT, si à la ligne 12 $level(u) = level(w)$ alors le cycle est impair et il est minimum. Sinon, il est pair et ne peut pas être minimum car la borne inférieur est de la forme $2k(v) + 1$.

Il faut encore regarder si il existe une arête (x, y) tel que $level(x) = level(y) = level(u)$. Le sommet x doit être soit un sommet encore dans la queue soit u lui-même. Lorsque FRONT(v) va terminer, on va calculer $F(v)$ tel que $F(v) = \{x\} \cup \{x | w \in V, x$ est dans la queue, $level(x) = level(u)\}$.

On va maintenant utiliser l'algorithme EDGE et $F(v)$ pour trouver une arête qui réponde aux conditions si-dessus.

Soit S un liste ordonnée de sommet distinct. On va introduire la notion de S-edge. (x, y) est un S-edge si x et y appartiennent à S. EDGE(S) va cherche une arête (u, w) dans S tel que $u < w$.

L'algorithme va utiliser le vecteur d'adjacence supérieur, pour un sommet v, on note $UA(v)$. C'est l'ensemble des sommets de v tel que pour un ordre donné, tous les éléments de $UA(v)$ sont supérieur à v.

Ce vecteur va être altéré durant l'algorithme, il faudra donc penser à utiliser une copie si on souhaite s'en servir plus tard.

L'algorithme fonctionne de la façon suivante :
- il commence par regarder si (u, w) n'est pas dans les $n^{1/3}$ derniers sommets de S (ligne 1 à 10).
- Pour cela, il regarde si les voisins du sommet i sont dans S (ligne 1 à 8).
- Si une arête est trouvée, elle est renvoyé par algorithme et il s'arrête (ligne 6).
- Si aucune arête n'est trouvée, alors on recherche de manière exhaustive dans les $n^{1/3}$ derniers sommets de S (ligne 11 à 19).
- Là encore, si une arête est trouvée, elle est renvoyé par algorithme et il s'arrête (ligne 16).
- Si aucune arête n'est trouvée, alors l'algorithme termine en renvoyant nil.

Nous allons maintenant voir comment utiliser FRONT et EDGE pour trouver un circuit minimum.

On va noter lmc la longueur du circuit minimum. L'algorithme MIN_CIRCUIT renvoie : lmc, qui est donc la longueur du circuit minimum, v, un sommet par lequel passe le circuit et a un sommet par lequel passe le circuit si il est impair.

L'idée de l'algorithme est de tester pour chaque $k(v)$ minimum si il est impair en cherchant un S-edge. Si aucun S-edge n'est trouver alors c'est que le cycle est pair.

L'algorithme fonctionne de la façon suivante :
- On commence par calculer k pour tous les sommets du graphe (ligne 1 à 3).
- On cherche ensuite le minimum parmi ces k (ligne 4).
- Si le k minimum est égal à l'infini (ligne 5), c'est qu'il n'y a pas de cycle dans le graphe et on arrête l'algorithme (ligne 7).

Algorithm 4 EDGE(S)

```
1: for i ← 1 step 1 until |S| − n^{1/3} do
2:    u ← S(i);
3:    while UA(u)isnotempty do
4:       Choisir aléatoirement un sommet w ∈ UA(u);
5:       if w ∈ S then
6:          return ((u, w));
7:       end if
8:       Supprimer w de UA(u);
9:    end while
10: end for
11: for i ← max(1, |S| − n^{1/3} + 1 step 1 until |S| do
12:    u ← S(i);
13:    for j ← i + 1 step 1 until |S| do
14:       w ← S(j);
15:       if (u, w) ∈ E then
16:          return ((u, w));
17:       end if
18:    end for
19: end for
20: return nil;
```

Algorithm 5 MIN_CIRCUIT(lmc, v, a)

```
1: for v ∈ V do
2:    FRONT(v);
3: end for
4: Trouver k_min;
5: if k_min = ∞ then
6:    lmc ← ∞
7:    return
8: end if
9: for v ∈ Vetk(v) = k_min do
10:   Trouver F(v);
11:   préparer une représentation de F(v) comme une liste chaînée triée;
12:   préparer un vecteur de bits de la représentation de F(v);
13:   α ← EDGE(F(v));
14:   if α ≠ nil then
15:      lmc ← 2k_min + 1;
16:      return
17:   end if
18: end for
19: lmc ← 2k_min + 2;
20: v ← tout sommet pour lesquels k est minimum;
```

- Si il y a bien un cycle, alors pour chaque sommet v du graphe tel que $k(v)$ est égal au minimum, on regarde si il est impair (ligne 9 à 18).
- Pour cela, on calcule $F(v)$ (ligne 10) et on applique EDGE dessus (ligne 13).
- Si une arête est trouvée alors le cycle est impair et $lmc = 2k_{min} + 1$ (ligne 15). On peut donc arreter l'algorithme (ligne 16).
- Sinon le cycle est pair et $lmc = 2k_{min} + 2$ (ligne 19).

MIN_CIRCUIT nous donne donc la longueur du circuit minimum, ainsi que un sommet v et une arête a.

Pour obtenir le cycle minimum, il suffit d'appliquer FRONT sur v si il est pair et dans le cas où le cycle est impair, il faut utiliser l'arête a pour clore le cycle.

5 Conclusion

La recherche de cycle est un vaste sujet. De nombreuses recherche ont été mené sur le sujet dans le but de toujours amélioré la complexité des algorithmes en fonction de ce que l'on cherche.

Nous avons pu voir quelques complexités et qu'elles ne sont pas forcément simple à comparer car, en plus de tenir compte des performances du pire cas, il faut prendre en compte les performances du temps moyen. Et qu'en plus de cela, il y a des algorithmes spécialisé pour tel ou tel conditions.

Nous avons ensuite vu 2 algorithmes. Le premier pour rechercher un trou de longueur k. Le second pour trouver un circuit minimum.

Références

[1] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3) :209–223, 1997.

[2] Maria Chudnovsky, Ken-ichi Kawarabayashi, and Paul Seymour. Detecting even holes. *J. Graph Theory*, 48 :85–111, February 2005.

[3] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM J. Comput.*, 7(4) :413–423, 1978.

[4] Stavros D. Nikolopoulos and Leonidas Palios. Hole and antihole detection in graphs. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 850–859, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[5] Raphael Yuster and Uri Zwick. Finding even cycles even faster. *SIAM J. Discret. Math.*, 10 :209–222, May 1997.